# TracingMyFacesUsingAspectJ

MyFaces is a very complex component implementation. Due to an aggressive use of the compositional pattern, it can often be difficult or obtuse to trace down a particular problem with specificity. Using MyFaces 1.1.1, this page will describe one possible mechanism for tracing the program logic of MyFaces at runtime using statically compiled AspectJ. More sophisticated applications of this concept, such as runtime weaving on JDK 1.5, are feasible extensions of this concept but the page will restrict itself to a basic compiled aspected trace and using such a trace to locate obscure failures within the MyFaces architecture.

Here is the aspect upon which the discussion will be based:

```
import java.io.BufferedWriter;
import java.io.File;
import java.io.FileWriter;
import java.io.PrintStream;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import com.dbz.lms.faces.action.DealHandler;
import org.apache.myfaces.lifecycle.LifecycleImpl;
import javax.faces.component.UIViewRoot;
import javax.faces.component.UIComponent;
import javax.faces.component.UIComponentBase;
import javax.faces.event.PhaseEvent;
import org.apache.myfaces.el.PropertyResolverImpl;
import javax.faces.context.FacesContext;
import org.apache.myfaces.context.servlet.ServletFacesContextImpl;
import javax.faces.application.NavigationHandler;
import org.apache.myfaces.application.ApplicationImpl;
import org.apache.myfaces.application.ActionListenerImpl;
import javax.faces.event.ActionEvent;
import javax.faces.event.ActionListener;
import javax.faces.component.UICommand;
import org.aspectj.lang.JoinPoint;

aspect Trace {
        private static final Log log = LogFactory.getLog(Trace.class);
        private static BufferedWriter fWrt;
        static {
                try {
                  fWrt = new BufferedWriter(new FileWriter(new File("C:\\trace.out")));
                } catch (Exception e) {
                        e.printStackTrace();
                }
       }

    pointcut myClass(Object obj): this(obj) &&
            (
                within(ApplicationImpl)
                || within(NavigationHandler)
                    || within(ServletFacesContextImpl)
                || within(LifecycleImpl)
                || within(UIViewRoot)
                || within(UIComponentBase)
                || within(UICommand)
                || within(ActionListenerImpl)
            )
            && !execution(UIComponentBase.new(..))
            && !execution(* ServletFacesContextImpl.getApplication(..))
            && !execution(* ServletFacesContextImpl.getExternalContext(..))
            && !execution(* ServletFacesContextImpl.getM*(..))
            && !execution(* ServletFacesContextImpl.getRenderKit(..))
            && !execution(* ServletFacesContextImpl.getResponseStream(..))
            && !execution(* ServletFacesContextImpl.getResponseWriter(..))
            && !execution(* ServletFacesContextImpl.getViewRoot(..))
            && !execution(* ServletFacesContextImpl.set*(..))
            && !execution(* UIComponentBase.findComponent(..))
            && !execution(* UIComponentBase.get*(..))
            && !execution(* UIComponentBase.set*(..))
            && !execution(* UIComponentBase.is*(..))
            && !execution(* UIComponentBase.save*(..))
```

```
        && !execution(* UIComponentBase.create*(..))
        && !execution(* UIComponentBase.encode*(..))
        && !execution(* UIComponentBase.processSaveState(..))
        && !execution(* UIComponentBase.restoreState(..))
        && !execution(* UIComponentBase.restoreValueBindingMap(..))
        && !execution(* UIComponentBase.restoreAttributesMap(..))
        && !execution(* UIViewRoot.getRenderKitId())
        && !execution(* UIViewRoot.getLocale())
        && !execution(* UIViewRoot.create*())
        && !execution(* ApplicationImpl.get*())
        && !execution(* ApplicationImpl.set*(..))
        && !execution(String *.toString())
        && !execution(String *.hashCode());

public static int TRACELEVEL = 2;
protected static PrintStream stream = System.out;
protected static int callDepth = 0;

public static void initStream(PrintStream s) {
    stream = s;
}

protected static void traceEntry(String str) {
    if (TRACELEVEL == 0) return;
    if (TRACELEVEL == 2) callDepth++;
    printEntering(str);
}

protected static void traceExit(String str) {
    if (TRACELEVEL == 0) return;
    printExiting(str);
    if (TRACELEVEL == 2) callDepth--;
}

private static void printEntering(String str) {
    String idt = getIndent();
    try {
        fWrt.write(idt + ">>>" + str + "\n");
        fWrt.flush();
    } catch (Exception e) {
        System.err.println(e.getMessage());
    }
}

private static void printExiting(String str) {
    String idt = getIndent();
    try {
        fWrt.write(idt + "<<<" + str + "\n");
        fWrt.flush();
    } catch (Exception e) {
              System.err.println(e.getMessage());
    }
}

private static void printIndent() {
    stream.print(getIndent());
}

private static String getIndent() {
        StringBuffer buf = new StringBuffer();
    for (int i = 0; i < callDepth; i++) {
              buf.append(" ");
    }
        return buf.toString();
}

pointcut myConstructor(Object obj): myClass(obj) && execution(new(..));
pointcut myMethod(Object obj): myClass(obj) &&
    (execution(* *.*(..))) && !execution(String toString());

before(Object obj): myConstructor(obj) {
```

```
        traceEntry(obj.hashCode() + ":" + thisJoinPointStaticPart.getSignature());
    }
    after(Object obj): myConstructor(obj) {
        traceExit(obj.hashCode() + ":" + thisJoinPointStaticPart.getSignature());
    }

    Object around(Object obj): myMethod(obj) {
            Object[] args = thisJoinPoint.getArgs();
            StringBuffer buf = new StringBuffer();
            buf.append(thisJoinPoint.getSignature());
            buf.append(" at line:" + thisJoinPoint.getSourceLocation().getLine());
            buf.append(": args[");
            for (int i=0; i<args.length; i++) {
                    if (i>0) {
                            buf.append(",");
                    }
                    buf.append(args[i] + "");
            }
            buf.append("]");
            Object thisObj = thisJoinPoint.getThis();
            String id = "";
            if (thisObj instanceof UIComponentBase) {
                    id = ((UIComponentBase)thisObj).getId();
            }
        traceEntry(obj.hashCode() + ":" + id + ":" + thisJoinPoint.getThis() + ":" + buf.toString());
        Object retVal = proceed(obj);
        traceExit(obj.hashCode() + ":" + thisJoinPointStaticPart.getSignature() + " returning " + retVal);
        return retVal;
    }
}
```

The development environment for following this example is as follows:

Eclipse 3.1.x from eclipse.org AJDT 1.3.1 from eclipse.org/ajdt

This page will not discuss the basic usage of the AspectJ Eclipse plugin but the AJDT site provides excellent tutorials:

http://www.eclipse.org/ajdt/demos/

Before using the above trace, please understand the basic workings of the AJDT so that you can convert your standard MyFaces Java project to use AspectJ.

Once you have Eclipse and the AJDT available, you will want to convert your standard Java project to an AspectJ project. Right click on the MyFaces project which you wish to debug and choose "Convert to AspectJ Project". Be aware that the AspectJ compiler may take a little longer to compile than the standard compiler, especially if you are aspecting a lot of project libraries.

Once the AspectJ nature is added, you will want to add the myfaces libraries to the AspectJ weaving path:

```
1) Right click on your MyFaces project
2) Choose "Properties" from the context menu
3) In the properties dialog, select the "AspectJ InPath" screen
4) On this screen choose the "Libraries and Folders" tab and add each of the MyFaces jars.  For MyFaces 1.1.1,
you should have three jars [possibly with different names]:
myfaces-api-1.1.1.jar, myfaces-impl-1.1.1.jar and tomahawk-myfaces-1.1.1.jar
5) Add the above trace source code to your project
6) The AspectJ compiler will now compile your source and will weave the trace into the MyFaces classes
7) Copy the ouput of the compilation to your web applications "WEB-INF/classes" directory and restart your
server
7a) NOTE: if you are not using a web application, you must merely ensure that the woven classes are first in
the runtime classpath of your application
```

Upon relaunching your application, once MyFaces actually runs, you should be able to "tail -f c:\trace.out" to follow the execution of MyFaces. Note that the file location should be changed as appropriate for Unix.

What is this trace doing? The trace is capturing the major classes upon which MyFaces processing depends so that the general application flow may be followed. You will notice that there are two major parts to the tracing pointcut: inclusive and exclusive. The inclusive parts:

```
            (
                within(ApplicationImpl)
                || within(NavigationHandler)
                    || within(ServletFacesContextImpl)
                || within(LifecycleImpl)
                || within(UIViewRoot)
                || within(UIComponentBase)
                || within(UICommand)
                || within(ActionListenerImpl)
            )
```

capture as much information as possible about the application flows. Each of the included classes plays a vital role in the lifecycle of MyFaces and piecing together what's happening when something goes wrong will invariably involve at least a few of the included classes.

The exclusive parts:

```
            && !execution(UIComponentBase.new(..))
            && !execution(* ServletFacesContextImpl.getApplication(..))
            && !execution(* ServletFacesContextImpl.getExternalContext(..))
            && !execution(* ServletFacesContextImpl.getM*(..))
//...
            && !execution(String *.toString())
            && !execution(String *.hashCode());
```

are designed to exclude cases where too much output is generated. The compositional approach taken by MyFaces results in a large number of similar invocations taking place which obscure the critial elements of application flow. Excluding the above cases produces a much cleaner trace. Note that in cases where too much output is straining the eyes, adding to the cases of exclusions will usually make tracing much easier.